On formal methods for certifying floating-point C programs

Ali Ayad

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

On formal methods for certifying floating-point C programs

Ali Ayad

N° 6927

Mai 2009

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--6927--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SACLAY - ÎLE-DE-FRANCE

On formal methods for certifying floating-point C programs

Ali Ayad*

Thème SYM — Systèmes symboliques
Équipe-Projet PROVAL

Rapport de recherche n° 6927 — Mai 2009 — 31 pages

Abstract: This paper presents an implementation of an extension of the *ACSL* specification language in the *Frama-C* tool in order to prove the correctness of floating-point C programs. This implementation is essentially based on the two *Why* models of floating-point arithmetic of [5]: the first model supposes that there is no overflow during the program execution and is called the *Real* model, i.e., the floating-point numbers are supposed to be reals and the floating-point operations are done with infinite precision. The second model checks that there is no overflow, i.e., proof obligations are generated by the *Why* tool to prove that the result of a floating-point operation is not greater than the maximal float allowed in the given type, this model is called the *Strict* model. In this paper, we describe a third model, called the *Full* model, which extends the *Strict* model. The *Full* model allows overflows and deals with special values: signed infinities, NaNs (Not-a-Number) and signed zeros as in the IEEE-754 Standard (see [2, 9]). The verification conditions generated by *Why* are (partially) proved by automatic theorem provers: Alt-Ergo, Simplify, Yices, Z3, CVC3 and Gappa or discharged in the interactive proof assistant Coq [20] using two existing Coq formalization of floating-point arithmetic: the first one from [12] and the second is the Gappa library [16]. When the *Why* proof obligations are written in the syntax of the Gappa library, we can use the *gappa* [3] and *interval* [15] tactics to achieve the proof. Several examples of floating-point C programs are presented in the paper to prove the efficiently of this implementation.

Key-words: static analysis of programs, formal methods, deductive verification, automatic theorem provers, interactive proof assistants, floating-point arithmetic, IEEE-754 standard

* This work was supported by the French national projects: *CerPan* (*Certification of numerical programs*, ANR-05-BLAN-0281-04), *Hisseo* (*Static and dynamic analysis of floating-point programs*, Digiteo 09/2008-08/2011) and *U3CAT* (*Unification of Critical C Code Analysis Techniques*, ANR-09-ARPEGE)

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université
4, rue Jacques Monod, 91893 ORSAY Cedex
Téléphone : +33 1 72 92 59 00

Sur les méthodes formelles de certification de programmes C flottants

Résumé : Ce papier présente une implémentation d'une extension du langage de spécification *ACSL* dans l'outil *Frama-C* dans le but de prouver la certitude de programmes C flottants. Cette implémentation est essentiellement basée sur les deux modèles *Why* de l'arithmétique flottante de [5]: le premier modèle suppose qu'il n'y a pas de dépassement de mémoire (overflow) durant l'exécution du programme et est appelé le modèle *Real*, i.e., les nombres flottants sont supposés de réels et les opérations flottantes sont faites avec une précision infinie. Le deuxième modèle vérifie qu'il n'y a pas d'overflows, i.e., des obligations de preuve sont générées par l'outil *Why* pour prouver que le résultat de l'opération flottante n'est pas plus grand que le flottant maximal du format de destination, ce modèle est appelé le modèle *Strict*. Dans ce papier, on décrit un troisième modèle, appelé le modèle *Full*, qui étend le modèle *Strict*. Le modèle *Full* permet d'avoir d'overflows et traite de valeurs spéciales: infinis signés, NaNs (Not-a-Number) et zéros signés comme dans la norme IEEE-754 (voir [2, 9]). Les conditions de vérification générées par *Why* sont (partiellement) prouvées par de prouveurs automatiques: Alt-Ergo, Simplify, Yices, Z3, CVC3 et Gappa ou par l'assistant de preuve Coq [20] en utilisant deux formalisations existantes de l'arithmétique flottante dans Coq: la première vient de l'article [12] et la deuxième est la bibliothèque Gappa [16]. Quand les obligations de preuve de *Why* sont écrites dans les syntaxes de la bibliothèque Gappa, on pourrait utiliser les deux tactiques Coq *gappa* [3] et *interval* [15] pour finir la preuve. Plusieurs exemples de programmes C flottants sont présentés dans ce papier pour prouver l'efficacité de cette implémentation.

Mots-clés : analyse statique de programmes, méthodes formelles, vérification déductive, prouveurs automatiques de théorèmes, assistants de preuve interactifs, arithmétique flottante, norme IEEE-754

1 Introduction

Many critical problems such as numerical analysis, physics, aeronautics (software embedded in satellites, robots), energy (nuclear centers), automotive, etc. deal with floating-point computations. As we rely more and more on softwares, the consequences of a bug are more and more dramatic, causing great financial and even human losses. We cite two historical accidents caused by bugs in softwares, the first one is the overflow bug that caused the failure of the Ariane 5 launcher in 1996: a double precision number which represents the horizontal speed of the launcher was converted into a 16 precision format, this speed was greater than the largest integer representable in 16 bits, the conversion failed without triggering an exception. The second one is the cumulated imprecision errors in a Patriot missile defense that caused it to miss its Scud missile target, resulting in 28 people being killed and around 100 injured in 1992 : the error was an inaccuracy in computing the date of the Patriot anti-missile. Thus for these reasons, one needs good tools to ensure the proper functioning of critical softwares.

There are many techniques to ensure that a program runs well every time we give it new input values and does not create errors at run-time such as division by zero and out-of-bound memory access. One can try to test all possible values of the input variables and verify that they have the correct outputs by the program. This can be done if the number of these tests is small and their running time is reasonable. *Dynamic analysis* performs tests to study important behaviors of the program. *Dynamic analysers* try to automatically generate path tests for programs and to extract properties that are true over all executions of the program. However, *static analyse* performs verification of programs without executing them by applying mathematical techniques such as first order logic and abstract interpretation.

The aim of this paper is to apply mathematical formal methods and theorem provers to static analysis of programs that handle floating-point arithmetic.

We are interested in certifying imperative programs, especially C programs. We will be able to prove the correctness of such a program, i.e., to ensure that the program is free of threats (null pointer dereferencing or out-of-bounds array access, etc ...) and to check that there is no overflow, never NaNs (Not-a-Number) when dealing with floating-point programs. Additionaly, one can prove some functional properties introduced manually as annotations in the source code. The general strategy is based on a classical Hoare logic framework [10] and follows the following steps:

Formal specification We annotate the program according to the properties we would to preserve after execution, i.e., preconditions, postconditions, (global, loop) invariants, termination, etc ... that determine the memory state of the imput and the output variables of the program. These annotations are generaly written in a programming language inspired from ML language. In this paper we use the ACSL language (ANSI C Specification Language [4]) which is partially implemented in the Frama-C verification tool [1].

Generation of verification conditions (i.e., logical formulas of first order that we should prove their validity). Different verification tools are intended to generate verification conditions from annotated C programs, e.g., the *Why* platform [6, 8] and its verification tool Caduceus [7] for C programs. In this paper, we will apply the *Frama-C* tool to an annotated C program which transforms it to an annotated program written in the *Jessie* language (an intermediate language for the verification of Java and C programs [13]). The *Jessie* plugin (see e.g., [18, 17] produces then the associated annotated *Why* file (with the syntaxes of the *Why* language). Finally, the *Why* platform generates the verification conditions that reflect the behavior of the C program specified by its annotations. All these transformations preserve the semantics of the annotated program. Note that *Why* does not interprete the annotations: they are built from functions and predicates axiomatization or defined in models that can be specific to each prover.

Formal verification These verification conditions are sent to a proof assistant in order to prove them. There are two kind of proof assistants: automated theorem provers such as *Simplify, Alt-Ergo, haRVey, Yices, Gappa*, etc ... and interactive proof assistants such as *Coq, PVS, Isabelle/HOL*, etc ...

This paper is organized as follows. Section 2 gives an overview of the floating-point arithmetic in the IEEE-754 standard [2]. In section 3, we describe the Frama-C compilation chain, i.e., the *ACSL, Jessie* and *Why* languages. Section 4 presents an improved version of the *Strict* model of [5] and a complete description of the *Full Why* model. A *Coq* formalization of these two models with an illustration of their efficiency on some examples are described also in Section 4. We terminate the paper by a conclusion with some perspectives in Section 5. Different tools and libraries are necessary to run correctly our examples, they are cited in a technical appendix.

2 Binary floating-point arithmetic

2.1 Binary floating-point representation

A binary floating-point number is a real number x represented by

$$x = (-1)^s \times m \times 2^e$$

where $s = 0$ or $s = 1$ (to determine the sign of x), $m \in \mathbb{N}$ is the significand and $e \in \mathbb{Z}$ is the exponent of x.

Definition 1 *A binary format f is a tuple (p, e_{min}, e_{max}) where p is a positive integer called the precision of f and e_{min} and e_{max} are two integers which define a range of exponents for f.*

Definition 2 *A binary floating-point number $x = (-1)^s \times m \times 2^e$ is representable in a binary format $f = (p, e_{min}, e_{max})$ if m and e verify the following inequalities :*

$$0 \leq m < 2^p \quad \textit{and} \quad e_{min} \leq e \leq e_{max}.$$

Of course, there is a finite number of binary floating-points representable in a given binary format f, i.e., f defines a finite subset of $\mathbb{R}$. The representation of a binary floating number x in f is not unique. For example, for $f = (3, -5, 6)$, $x = 6 \times 2^0 = 3 \times 2$ has two different binary representations in f. In general, real numbers with denominator not a power of 2 are not representable in any binary formats, they are approximated by a representable binary floating-point number w.r.t. a certain rounding direction (see below). For example, for $f = (24, -149, +104)$, 0.1 is approximated by the binary floating-point number 13421773×2^{-27} which is representable in the format f.

2.2 The IEEE-754 standard

The standard IEEE-754-2008 [2] defines methods to perform binary and decimal floating-point arithmetic. It facilitates the portability of programs between computers which support the standard i.e., programs will product the same results of numerical computations on different architectures that implement the standard. The standard specifies binary and decimal formats to represent floating-point numbers with the operations of conversion between these formats. It specifies the elementary operations (addition, subtraction, multiplication, division, square root, Fused-Multiply-Add, remainder, etc ...) and the comparison operators on floating-point numbers. It specifies conversion operators between different formats and between integers and floating-point numbers. It handles floating-point exceptions, including data that are not numbers (NaNs).

2.2.1 Formats

The standard has introduced three basic binary formats: **binary32** (`Single`), **binary64** (`Double`) and **binary128** (`Quad`) which are defined with the above notations by:

- **binary32** = (24, -149, +104)
- **binary64** = (53, -1074, +971)

- **binary128** = (113, -16494, +16271)

For example, a real number representable in **binary32** is encoded by a string of bits of length 32 (1 bit for the sign, 23 bits to represent the significand and 8 bits for the exponent).
The standard defines also two decimal formats **decimal64** and **decimal128**, extended and extendable binary and decimal formats.

2.2.2 Normal and denormal binary floating-point numbers

Let $f = (p, e_{min}, e_{max})$ be a binary format and $x = (-1)^s \times m \times 2^e$ be a binary floating-point number which is representable in f. In this subsection, we define normal, denormal and canonic binary floating-point numbers as follows:

Definition 3 *The binary floating number x is said to be normalized in the format f if $2 \times m \geq 2^p$, this is equivalent to that the leading bit in the binary representation of the significand m is nonzero, i.e., m is represented by exactly p bits.*

Definition 4 *The binary floating number x is said to be denormalized in the format f if the exponent e of x is minimal and $2 \times m < 2^p$, this is equivalent to that $e = e_{min}$ and the leading bit in the binary representation of the significand m is zero, i.e, m is represented by fewer than p bits.*

Definition 5 *The binary floating number x is said to be canonic in the format f if it is normal or denormal in f.*

The smallest positive normalized floating-point number in the format f is $2^{e_{min}+p-1}$ and the largest is $2^{e_{max}}(2^p - 1)$. The smallest positive denormalized floating number in the format f is $2^{e_{min}}$. In the standard, zero is neither normalized nor denormalized and there are distinct representations for $+0$ and -0 for all formats but $+0 = -0$.

2.2.3 Rounding direction and basic operations

The standard requires that every floating-point operation shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range of exponent, and then rounded that result according to the correspond rounding direction. Rounding a real number x (i.e., a result of an operation with infinite precision) modifies it to fit in the destination's format f of the operation. The standard has defined five rounding directions as follows: one can always find two representable floating-point numbers x_1 and x_2 in the format f such that $x_1 \leq x \leq x_2$ (if x is representable in f then $x_1 = x_2 = x$).

1. **roundTiesToEven** (`nearest_to_even`): the rounding of x w.r.t. this direction is the nearest floating-point number to x among x_1 and x_2. If x is in the middle of the interval $[x_1, x_2]$ then one chooses whose significand is even.

2. **roundTowardPositive** (`up`): the rounding of x is x_2.

3. **roundTowardNegative** (`down`): it is the opposite of **roundToward-Positive**, i.e., the rounding of x is x_1.

4. **roundTowardZero** (`to_zero`): the rounding of x w.r.t. this direction is x_1 if $x > 0$ and x_2 if $x < 0$.

5. **roundTiesToAway** (`nearest_away`): the rounding of x is the closest to x among x_1 and x_2. If x is exactly the middle of the interval $[x_1, x_2]$ then the rounding is x_2 if $x > 0$ and x_1 if $x < 0$.

If the infinitely precise sum (or minus) of two floating-point numbers is zero, then the result is -0 if the rounding direction is **roundTowardNegative** and $+0$ otherwise. The standard classified floating-point operations in two categories:

- **Homogeneous operation:** the operands and the result have the same format in the same base.
- **Non-homogeneous operation:** the operands and the result have different formats in the same base.

2.2.4 Special values and exceptions

The standard defined three special values: $-\infty, +\infty$ and NaN (Not-a-Number). These numbers should be treated both in the input and the output of the arithmetic operations as usually. For example, $(+\infty) + (+\infty) = (+\infty)$, $1/(+\infty) = +0$, $1/(-0) = -\infty$, $(+0)/(+0) = NaN$, $(-0) \times (+\infty) = NaN$ and $\sqrt{-1} = NaN$ but $\sqrt{-0} = -0$. Operations propagate NaN operands to their results, then NaNs are not rounded. In addition, the standard handles five exceptions:

Invalid operation exception It holds when the operation is not defined on the operands. For example, the square root of negative numbers and the division of two zeros. The result of an invalid operation is NaN.

Inexact operation exception It holds when the result of the operation is not representable in the destination's format. This result must be rounded.

Division by zero exception It happens in the case of division of a nonzero and non NaN number by zero. The result of this division is infinity with sign the product of the signs of the operands.

Overflow Overflow is said to occur when the exact result (i.e., when the exponent range unbounded) of a floating-point operation is finite but with an absolute value that is larger than the maximal floating-point number represented in the destination's format. As with division by zero, in the days before IEEE arithmetic was available the usual treatment of overflow was to set the result to ($\pm$) the largest floating-point number in the destination's format or to interrupt or terminate the program. The IEEE-754 standard response to overflow is to deliver the correctly rounded result, either ($\pm$) the maximal floating-point number or $\pm\infty$. The result of an overflow operation depends on the rounding direction and the sign of the intermediate result as follows: in the case of **roundTiesToEven** and **roundTiesToAway**, this is the same as saying that overflow occurs when an exact finite result is rounded to $\pm\infty$ according to the sign of the result, but it is not the same for the other rounding modes. If the rounding direction is **roundTowardPositive**, the result is $+\infty$ (if it is positive) or the opposite

of the largest normalized floating-point number of the destination's format (if it is negative). For **roundTowardNegative**, the result is the largest normalized floating-point number of the destination's format (if it is positive) or $-\infty$ (if it is negative). **roundTowardZero** is analogue to **roundTowardNegative** if the result is positive and to **roundTowardPositive** if the result is negative.

Underflow Underflow is said to occur when the exact result of an operation is nonzero but with an absolute value that is smaller than the smallest normalized floating-point number represented in the destination's format. In the days before IEEE arithmetic, the response to underflow was typically, though not always, `flush to zero`, i.e., return the result 0. The IEEE-754 standard response to underflow is to return the correctly rounded value, which may be a denormal number, ± 0 or ($\pm$) the minimal denormal number. This is known as `gradual underflow`.

2.3 Rounding errors

One of the major problems encountered in floating-point computations is the propagation of rounding errors. Because of the fact that not all floating-point operations are exacts, the final result of a chain of floating-point instructions can be very far from the exact result (i.e., the expected result if the computation is done on real numbers). There are three concepts of measurement to bound the errors in operations: the `absolute error`, the `relative error` and the `ulp error` (unit in the last place). If $x = (-1)^s \times m \times 2^e$ is the result of a floating-point operation rounded in the destination's binary format $f = (p, e_{min}, e_{max})$ of the operation w.r.t. a certain rounding direction.

- The `absolute error` of x is the distance between x and the exact result of the operation (i.e., without rounding).
- The `relative error` ϵ of x is the division of the `absolute error` by the exact result y of the operation, i.e., $x = (1 + \epsilon)y$. One can bound this error by $|\epsilon| \leq 2^{-p}$ if the rounding direction is **roundTiesToEven** and by $|\epsilon| \leq 2^{1-p}$ otherwise.
- $\texttt{ulp}(x)$ is the weight of the last bit in the significand m of x, i.e., $\texttt{ulp}(x) = 2^p$. For a real r, $\texttt{ulp}(r) = 2^{\lfloor log_2 |r| \rfloor + 1 - p}$.

3 The intermediate languages in the Frama-C chain

3.1 The ACSL language

ACSL is a specification language of C programs. It is inspired from the specification language of the *Caduceus* tool and the Java Modeling Language (JML). A part of *ACSL* is implemented in the *Frama-C* framework, another part is under experimentation. In this paper, we extend the syntaxes of *ACSL* to handle floating-point computations. Annotations in C programs are written in comments that start by `/*@` or `//@` and end as in C comments. In the sequel, we show only a brief overview of the *ACSL* language, people interested in details can return to the tutorial [19] or the reference manual [4].

3.1.1 Types in ACSL

- The types `integer`, `real` and `boolean` represent respectively mathematical integers, real numbers and booleans (with just two values `\true` and `\false`).
- Users can introduce new logic types in *ACSL*.
- C integral types `char`, `short`, `int` and `long`, `signed` or `unsigned`, are all subtypes of the type `integer` which is itself a subtype of the `real` type.

3.1.2 Annotations of C functions

There are built-in constructs that evaluate the pre-state and the post-state of a C function. The keywords `\old(e)` and `\result` are reserved in *ACSL* to denote the value of the expression e before the execution of the function and the value returned by the function respectively. Simple contracts are written immediately before the function field.

- For a set P of predicates, the contract `requires P;` means that the predicates of P are satisfied by the input variables of the function.
- For a set L of memory locations, the contract `assigns L;` enforces that no memory locations outside L can be modified. It does not enforce modifications of locations in L.
- For a set Q of predicates, the contract `ensures Q;` asserts that the predicates of Q hold on the output variables of the function.

3.1.3 Annotations of C loops

They are written immediately before the C loops *for, while*, etc ...

- For a set I of inductive predicates, the contract `loop invariant I;` means that the predicates of I are satisfied at every loop iteration.
- For a set L of memory locations, the contract `loop assigns L;` has the same meaning as `assigns L;` for every loop iteration.

- For an integer expression e, the contract `loop variant e;` ensures that the loop terminates after a finite number of steps if e decreases during the loop iterations

3.1.4 Global properties

- Global invariants: for a set P of predicates, the contract `global invariant P;` means that the predicates of P are always true on global variables of the program.
- Type invariants: for a type τ and a set P of predicates defined over τ, the contract `type invariant P;` means that the predicates of P are satisfied by any variable of type τ.
- Logic specifications: users of *ACSL* are able to introduce new logic types, logic definitions, functions, predicates and axioms.

3.2 The Jessie intermediate language

Jessie language is an intermediate language presented in [13]. It provides primitive types such as integers, booleans, reals, etc ... and abstract datatypes. *Jessie* has not a notion of arrays, structures, pointers and memory heap. *Jessie* programs can be annotated using pre- and postconditions, loop invariants and intermediate assertions. Logical formulas are written in a typed first-order logic with built-in equality, booleans, infinite precision integers and real numbers with usual operations. We have extended this language by intoducing two new types, `float` and `double` as in C programs to deal with floating-point computations. Ordinary operations and comparison over `float` and `double` variables are defined. We have introduced rounding modes and casting operators to *Jessie*. Special predicates are devoted to test if a `float` or `double` variable is finite, NaN or infinity.

3.3 The Why language

The platform Why aims at being a verification conditions generator (VCG) back-end for other verification tools. It provides a powerful input language including higher-order functions, polymorphism, references, arrays and exceptions.

The *Why* language is inspired from the *Ocaml* [11] language for functionnal programs. So there is no distinction between instructions and expressions. Special keywords are reserved in *Why*, for example, `result` denotes the value returned by a function and is used only in the postcondition. Programs in *Why* language can be annotated by preconditions, assertions, postconditions, loop invariant and loop variant to ensure the termination. The language contains basic built-in types: `bool`, `unit`, `int` and `real`. The expressions in *Why* are formed by:

- The integer, boolean constants.
- The variables and the function call `f e_1 ... e_n`.

- The basic unary and binary operations.
- The local definition `let x = e_1 in e_2` and the sequence of instructions `let _ = e_1 in e_2`.
- The local reference `let x = ref e_1 in e_2`.
- The dereferenciation `!x`, the assigment `x := e`.
- The conditionnal `if e_1 then e_2 else e_3`.
- The loop `while e_1 do e_2 done`.
- The function declaration `fun (x_1 : t_1) ... (x_n : t_n) -> e`.
- The recursive function `let rec f (x_1 : t_1,...,x_n : t_n) in e`.

The *Why* tool implements this programming language. It takes annotated programs as input and generates proof obligations for a wide set of interactive proof assistants (Coq, PVS, Isabelle/HOL, HOL 4, HOL-Light, Mizar) and decision procedures (Simplify, Ergo, Yices, CVC Lite, CVC3, haRVey, Zenon). It can verify algorithms rather than programs, since it implements a rather abstract and idealistic programming language. Several non-trivial algorithms have already been verified using the Why tool, such as the Knuth-Morris-Pratt string searching algorithm for instance. It computes Dijkstra's weakest preconditions in the Floyd-Hoare logic. The *Why* tool can be used to write axiomatizations and goals and to dispatch them to several existing provers. It is independent of the back-end prover that will be used, it makes no assumption regarding the logic used. It uses a syntax of polymorphic first-order predicate logic for annotations with no particular interpretation. Function symbols and predicates can be declared in order to be used in annotations, together with axioms, and they may be given definitions on the prover side later, if needed.
For more details on the *Why* language and tool, you can refer to the reference manual [6, 8]. The extension to handle floating-point programs is given in the next section.

4 Why models for floating-point C programs

In this section, we describe an improved version of the *Strict Why* model of [5] where any operation on floating-point numbers produces a proof obligation to ensure that there is no overflow. We introduce also the *Full Why* model, i.e., the complete model which implements the IEEE-754 standard, e.g., infinities, signed zeros and NaNs are supposed to be special values in the program. We give an implementation of these two models in the *Coq* proof assistant with the syntax of the Gappa librabry. Some examples will be treated at the end of this section.

4.1 The Strict model

This model is based on the *Why* theory and written in the syntaxe of the *Why* language. In [5], a floating-point value is viewed as three parts:

1. the floating-point number of the given type, as computed by the program;
2. a real number, which would be the value if all previous computations were performed on real numbers and thus exact;
3. another real number, which is the value which should be ideally computed.

We define new types, constants, functions and predicates to be able to write annotated floating-point *Why* programs. The complete description of this model can be found in the next *Why* distribution in the file `lib/why/floats_strict.why`.

In the logic:

- An abstract type `gen_float` for generic floats.
- A type `float_format` with three elements: `Single`, `Double` and `Quad`.
- A type `mode` for the five rounding modes of the revision of the IEEE-754 standard: `nearest_even, to_zero, up, down, nearest_away` as described in Section 2.
- Functions `float_value, exact_value, model_value` to access respectively to the floating-point, the exact and the model part of variables of type `gen_float`.
- A rounding operator `gen_float_of_real_logic` which rounds a real to a `gen_float` according to a certain float format and rounding direction.
- A function `round_float` which is the rounding operator of reals with unbounded range of exponents.
- A function `max_gen_float` which is the maximal real representable number for each `float_format`. Its values are axiomatized by (constants are written in hexadecimal format):

```
axiom max_single: max_gen_float(Single) = 0x1.FFFFFEp127
axiom max_double: max_gen_float(Double) =
                    0x1.FFFFFFFFFFFFFp1023
axiom max_quad: max_gen_float(Quad) =
                    0x1.FFFFFFFFFFFFFFFFFFFFFFFFFFFFp16383
```

- A predicate `no_overflow` which indicates that there is no overflow when the result of an operation is computed with infinite precision and unbounded range of exponents. It is defined by

```
predicate no_overflow(f:float_format, m:mode, x:real) =
    abs_real(round_float(f,m,x)) <= max_gen_float(f)
```

- Functions to handle rounding errors: `gen_round_error` for the absolute error, `gen_relative_error` for the relative error and `gen_total_error` for the total error. They are defined by

```
function gen_round_error(x:gen_float) : real =
    abs_real(float_value(x) - exact_value(x))

function gen_relative_error(x:gen_float) : real =
    abs_real(float_value(x) - exact_value(x))/exact_value(x)

function gen_total_error(x:gen_float) : real =
    abs_real(float_value(x) - model_value(x))
```

In the program:

- A rounding operator `gen_float_of_real` which rounds a real to a `gen_float` according to a certain float format and rounding direction.
- Basic binary and unary operations: `add_gen_float`, `sub_gen_float`, `mul_gen_float`, `div_gen_float`, `sqrt_gen_float`, `neg_gen_float`, `abs_gen_float` on the `gen_float` type.
- Basic operators for comparison: `lt_gen_float`,`gt_gen_float`, etc ...
- A casting operator `cast_gen_float` between float formats according to rounding directions.

In this model, pre- and postconditions are written for each basic operation of the program. These conditions product proof obligations for testing that there is no overflow at run-time. For example, multiplication is specialized by:

```
parameter mul_gen_float :
   f:float_format -> m:mode -> x:gen_float -> y:gen_float ->
   { no_overflow(f,m,float_value(x) * float_value(y)) }
   gen_float
   { float_value(result) =
          round_float(f,m,float_value(x) * float_value(y))
     and
     exact_value(result) = exact_value(x) * exact_value(y)
```

```
     and
     model_value(result) = model_value(x) * model_value(y)
   }
```

In some particular functions, like division and square root, this gives conditions on their parameters. For example, `div_gen_float` is parametrized by:

```
parameter div_gen_float : f:float_format -> m:mode ->
    x:gen_float -> y:gen_float ->
    { float_value(y) <> 0.0
      and
      no_overflow(f,m,float_value(x)/float_value(y))
    }
    gen_float
    { float_value(result) =
           round_float(f,m,float_value(x)/float_value(y))
      and
      exact_value(result) = exact_value(x)/exact_value(y)
      and
      model_value(result) = model_value(x)/model_value(y)
    }
```

Comparisons between two `gen_float` variables x and y is equivalent to the comparison of their associated float values with the mathematical sense. For example, `lt_gen_float` is specialized by:

```
parameter lt_gen_float: x:gen_float -> y:gen_float ->
    {}
    bool
    { if result then float_value(x) < float_value(y)
      else float_value(x) >= float_value(y)
    }
```

The rounding and the cast operators are specified by:

```
parameter gen_float_of_real :
    f:float_format -> m:mode -> x:real ->
    { no_overflow(f,m,x) }
    gen_float
    { float_value(result) = round_float(f,m,x)
      and
      exact_value(result) = x
      and
      model_value(result) = x
    }

parameter cast_gen_float :
      f:float_format -> m:mode -> x:gen_float ->
    { no_overflow(f,m,float_value(x)) }
    gen_float
```

```
{ float_value(result) = round_float(f,m,float_value(x))
  and
  exact_value(result) = exact_value(x)
  and
  model_value(result) = model_value(x)
}
```

4.1.1 Floating-point arithmetic in ACSL with the Strict model

All the above definitions are passed to the *ACSL* language to be able to write annotated C programs. They are defined as follows:

- Constants: floating-point constants are interpreted as mathematical real numbers in the specifications, they can be written in decimal or in hexadecimal format. Integers are also seen as mathematical real numbers if necessary. Variables of type `float` or `double` in the C program are implicitly converted to the `real` type when used in the annotations.
- Casts: a built-in logic type is defined by the five `rounding_mode` of the IEEE-754 standard: `\NearestEven`, `\ToZero`, `\Up`, `\Down`, `\NearestAway`. Casting from a C `integer` type or a `float` type to a `float` or a `double` is as in C. Conversion of real numbers to `float` or `double` values is given by the following two logic functions:

  ```
  logic float \round_float(rounding_mode m, real x);
  logic double \round_double(rounding_mode m, real x);
  ```

  Cast operators (`float`) and (`double`) applied to a mathematical integer or real number x are equivalent to apply the above rounding functions with the `\NearestEven rounding_mode` (which is the default rounding mode in C programs). Suffixes f and l are meaningful, because they implicitly add a cast operator as above.
- Floating-point operations: in the *Strict* model, the sum (resp. minus, product and division) of two C floating-point variables x and y is equivalent to the sum (resp. minus, product and division) of their associated real numbers with the mathematical sense.
- A special predicate to verify if there is an overflow in floating-point operations is declared by:

  ```
  predicate \no_overflow(f:float_format,rounding_mode m,real x);
  ```

  where `float_format` is a built-in logic type with three elements: `\Single`, `\Double` and `\Quad`.
- Rounding errors: according to the *Strict* model, a floating-point expression e is represented by three parts: the floating-point part, i.e, the value computed by the program, the exact part which is a real value accessed by the construct `\exact(e)` and the model part, a real value accessed by the constructs `\model(e)`. The rounding errors are computed by the construct

`\round_error(e)` which is the difference between the floating-point part and the exact part of e, the construct `\total_error(e)` which is the difference between the floating-point part and the model part of e and the construct `\relative_error(e)` which is the quotient of `\round_error(e)` by `\exact(e)`.

4.1.2 A coq formalization of the Strict model

After the specification of the *Strict* model in the *Why* syntaxes, the *Why* tool can now generate verification conditions for programs according to specific annotations. The automatic and interactive provers supported by the *Why* platform are not until now able to prove these properties. One has to translate our model to them with their syntaxes. We have choose to formalize our models in the *Coq* proof assistant by using an existing implementation of floating-point arithmetic, the *Gappa* library [16]. *Why* translates automaticaly definitions and predicates from its language to that of *Coq*.

The `gen_float` type is defined to be the type `float2` of the *Gappa* library, i.e., the set of couples of integers. The floating-point value is then given by

```
Definition float_value x := float2R x.
```

The `round_float` operator for rounding reals to floating-point numbers is defined using the functions `gappa_rounding` and `rounding_float` as follows:

```
Definition round_float (f : float_format) (m : mode) (x:R) :=
match f with
 | Single =>
     gappa_rounding (rounding_float (round_mode m) 24 149) x
 | Double =>
     gappa_rounding (rounding_float (round_mode m) 53 1074) x
end.
```

where `R` is the type of reals in *Coq* and `round_mode` is a kind of translate function to the rounding modes of *Gappa*.

4.2 The Full model

The *Full* model allows overflows, NaNs, infinities and signed zeros. In this model, a floating-point number is still viewed with three parts: the real, exact and model parts. The complete description of this model can be found in the next *Why* distribution in the file `lib/why/floats_full.why`.

In the logic: We keep the same types (`gen_float`, `float_format`, `mode`) and functions (`float_value`, `exact_value`, `mode_value`) as in the *Strict* model.

In addition, we introduce two new types: the type `Float_class` with three elements `Finite`, `Infinite` and `NaN` and the type `sign` with two elments `Negative` and `Positive`. Two additional functions `float_class` and `float_sign` are necessary to indicate respectively the class and the sign of a `gen_float`. Special predicates are defined to test if a `gen_float` variable is finite, NaN, infinities

or signed zero: `is_finite`, `is_NaN`, `is_plus_infinity`, `is_minus_infinity`, `is_gen_zero`, `is_gen_zero_plus` and `is_gen_zero_minus`. They are naturally defined by:

```
predicate is_finite(x:gen_float) =
   float_class(x) = Finite
predicate is_infinite(x:gen_float) =
   float_class(x) = Infinite
predicate is_NaN(x:gen_float) =
   float_class(x) = NaN
predicate is_minus_infinity(x:gen_float) =
   is_infinite(x) and float_sign(x) = Negative
predicate is_plus_infinity(x:gen_float) =
   is_infinite(x) and float_sign(x) = Positive
predicate is_gen_zero(x:gen_float)=
   is_finite(x) and float_value(x) = 0.0
predicate is_gen_zero_plus(x:gen_float) =
   is_gen_zero(x) and float_sign(x) = Positive
predicate is_gen_zero_minus(x:gen_float) =
   is_gen_zero(x) and float_sign(x) = Negative
```

Comparison between two `gen_float` numbers is given by the predicates `float_le_float`, `float_lt_float`, `float_eq_float` and `float_ne_float`:

```
predicate float_le_float(x:gen_float,y:gen_float) =
   (is_finite(x) and is_finite(y)
                 and float_value(x) <= float_value(y))
    or (is_minus_infinity(x) and not is_NaN(y))
    or (not is_NaN(x) and is_plus_infinity(y))

predicate float_lt_float(x:gen_float,y:gen_float) =
   (is_finite(x) and is_finite(y)
                 and float_value(x) < float_value(y))
    or (is_minus_infinity(x) and not is_NaN(y)
                 and not is_minus_infinity(y))
    or (not is_NaN(x) and not is_plus_infinity(x)
                 and is_plus_infinity(y))

predicate float_eq_float(x:gen_float,y:gen_float) =
   not is_NaN(x) and not is_NaN(y) and
   ((is_finite(x) and is_finite(y)
                  and float_value(x) = float_value(y))
     or
   (is_infinite(x) and is_infinite(y) and same_sign(x,y)))

predicate float_ne_float(x:gen_float,y:gen_float) =
   not float_eq_float(x,y)
```

where `same_sign` is defined by:

```
predicate same_sign(x:gen_float,y:gen_float) =
   float_sign(x) = float_sign(y)
```

In the program: In the full model there are no preconditions on the unary and binary operations. If the result of a floating-point operation is finite and nonzero, the axiom `finite_sign` ensures that the sign of this result is the same as that of its floating value (using a predicate `same_sign_real`):

```
axiom finite_sign : forall x:gen_float.
   (is_finite(x) and float_value(x) <> 0.0) ->
    same_sign_real(x,float_value(x))
```

and if the result is zero and the operation is an addition or a subtraction, the predicate `sign_zero_resut` indicates the `sign` of the result according to the rounding direction as it is specified in the IEEE-745 standard:

```
predicate sign_zero_result(m:mode,x:gen_float) =
   float_value(x) = 0.0 ->
   ((m = down -> float_sign(x) = Negative)
     and
    (m <> down -> float_sign(x) = Positive))
```

If the result is greater than the maximal floating-point number representable in the destination's format, its class depends on its sign and the current rounding direction as described in Section 2 (paragraph on overflows). This is given by the predicate `overflow_value` which is defined by:

```
predicate overflow_value(f:float_format,m:mode,x:gen_float) =
   (m = down ->
        (float_sign(x) = Negative -> is_infinite(x)) and
        (float_sign(x) = Positive -> is_finite(x) and
                      float_value(x) = max_gen_float(f)))
   and
   (m = up ->
        (float_sign(x) = Negative -> is_finite(x) and
                      float_value(x) = - max_gen_float(f)) and
        (float_sign(x) = Positive -> is_infinite(x)))
   and
   (m = to_zero -> is_finite(x) and
        (float_sign(x) = Negative ->
                      float_value(x) = - max_gen_float(f)) and
        (float_sign(x) = Positive ->
                      float_value(x) =   max_gen_float(f)))
   and
   (m = nearest_away or m = nearest_even -> is_infinite(x))
```

The rounding operator `gen_float_of_real`, the basic binary and unary operations, the basic operators for comparison and the casting operator `cast_gen_float` between float formats are specialized according to the *Full* model. Here we show the complete specification of some of them:

```
parameter gen_float_of_real:
    f:float_format -> m:mode -> x:real ->
    { }
    gen_float
```

```
    { no_overflow(f,m,x) -> is_finite(result) and
                    float_value(result) = round_float(f,m,x)
      and
      not no_overflow(f,m,x) -> same_sign_real(result,x)
                              and overflow_value(f,m,result)
      and
      exact_value(result) = x
      and
      model_value(result) = x
    }

parameter add_gen_float :
    f:float_format -> m:mode -> x:gen_float -> y:gen_float ->
    { }
    gen_float
    { (is_NaN(x) or is_NaN(y) -> is_NaN(result))
    and
      ((is_finite(x) and is_infinite(y)) -> (is_infinite(result)
              and same_sign(result,y)))
    and
      ((is_infinite(x) and is_finite(y)) -> (is_infinite(result)
              and same_sign(result,x)))
    and
      ((is_infinite(x) and is_infinite(y) and same_sign(x,y)) ->
             (is_infinite(result) and same_sign(result,x)))
    and
      ((is_infinite(x) and is_infinite(y) and diff_sign(x,y)) ->
              is_NaN(result))
    and
      ((is_finite(x) and is_finite(y) and
        no_overflow(f,m,float_value(x)+float_value(y))) ->
             (is_finite(result) and float_value(result) =
              round_float(f,m,float_value(x)+float_value(y))
              and sign_zero_result(m,result)))
    and
       ((is_finite(x) and is_finite(y) and
         not no_overflow(f,m,float_value(x)+float_value(y))) ->
             (same_sign_real(result,float_value(x)+float_value(y))
              and overflow_value(f,m,result)))
    and
        exact_value(result) = exact_value(x) + exact_value(y)
    and
        model_value(result) = model_value(x) + model_value(y)
    }
```

where `diff_sign` is defined by:

```
predicate diff_sign(x:gen_float,y:gen_float) =
   float_sign(x) <> float_sign(y)

parameter mul_gen_float :
```

```
    f:float_format -> m:mode -> x:gen_float -> y:gen_float ->
    { }
    gen_float
    { ((is_NaN(x)  or  is_NaN(y)) -> is_NaN(result))
    and
      ((is_gen_zero(x) and is_infinite(y)) -> is_NaN(result))
    and
      ((is_finite(x) and is_infinite(y) and
           float_value(x) <> 0.0) -> is_infinite(result))
    and
      ((is_infinite(x) and is_gen_zero(y)) -> is_NaN(result))
    and
      ((is_infinite(x) and is_finite(y) and
           float_value(y) <> 0.0) -> is_infinite(result))
    and
      ((is_infinite(x) and is_infinite(y)) -> is_infinite(result))
    and
      ((is_finite(x) and is_finite(y) and
        no_overflow(f,m,float_value(x)*float_value(y))) ->
           (is_finite(result) and
            float_value(result) =
            round_float(f,m,float_value(x) * float_value(y))))
    and
      ((is_finite(x) and is_finite(y) and
        not no_overflow(f,m,float_value(x)*float_value(y))) ->
                  (overflow_value(f,m,result)))
    and
        product_sign(result,x,y)
    and
        exact_value(result) = exact_value(x) * exact_value(y)
    and
        model_value(result) = model_value(x) * model_value(y)
    }
```

where `product_sign` is defined by:

```
predicate product_sign(z:gen_float,x:gen_float,y:gen_float) =
   (same_sign(x,y) -> float_sign(z)= Positive) and
   (diff_sign(x,y) -> float_sign(z)= Negative)
```

```
parameter div_gen_float :
    f:float_format -> m:mode -> x:gen_float -> y:gen_float ->
    { }
    gen_float
    { ((is_NaN(x) or is_NaN(y)) -> is_NaN(result))
    and
      ((is_finite(x) and is_infinite(y)) -> is_gen_zero(result))
    and
      ((is_infinite(x) and is_finite(y)) -> is_infinite(result))
    and
      ((is_infinite(x) and is_infinite(y)) -> is_NaN(result))
```

```
    and
      ((is_finite(x) and is_finite(y) and float_value(y) <> 0.0
        and no_overflow(f,m,float_value(x)/float_value(y))) ->
           (is_finite(result) and float_value(result) =
               round_float(f,m,float_value(x)/float_value(y))))
    and
      ((is_finite(x) and is_finite(y) and float_value(y) <> 0.0
        and not no_overflow(f,m,float_value(x)/float_value(y)))
            -> overflow_value(f,m,result))
    and
      ((is_finite(x) and is_gen_zero(y) and
        float_value(x) <> 0.0) -> is_infinite(result))
    and
      ((is_gen_zero(x) and is_gen_zero(y)) -> is_NaN(result))
    and
      product_sign(result,x,y)
    and
      exact_value(result) = exact_value(x)/exact_value(y)
    and
      model_value(result) = model_value(x)/model_value(y)
    }

parameter cast_gen_float :
    f:float_format -> m:mode -> x:gen_float ->
    { }
    gen_float
    { (is_NaN(x) -> is_NaN(result))
    and
      (is_infinite(x) -> (is_infinite(result) and
           same_sign(result,x)))
    and
      ((is_finite(x) and no_overflow(f,m,float_value(x))) ->
          (float_class(result) = Finite and
           float_value(result) = round_float(f,m,float_value(x))))
    and
      ((is_finite(x) and not no_overflow(f,m,float_value(x))) ->
           same_sign(result,x) and overflow_value(f,m,result))
    and
       exact_value(result) = exact_value(x)
    and
       model_value(result) = model_value(x)
    }

parameter lt_gen_float : x:gen_float -> y:gen_float ->
    { }
    bool
    { ((is_NaN(x) or is_NaN(y)) -> result = false)
    and
      ((is_finite(x) and is_infinite(y)) ->
           if result then float_sign(y) = Positive
```

```
        else float_sign(y) = Negative)
  and
    ((is_infinite(x) and is_finite(y)) ->
        if result then float_sign(x) = Negative
        else float_sign(x) = Positive)
  and
    ((is_infinite(x) and is_infinite(y)) ->
        if result then (float_sign(x) = Negative
           and float_sign(y) = Positive)
        else (float_sign(x) = Positive
              or float_sign(y) = Negative))
  and
    ((is_finite(x) and is_finite(y)) ->
        if result then float_value(x) < float_value(y)
        else float_value(x) >= float_value(y))
  }
```

4.2.1 Floating-point arithmetic in ACSL with the Full model

In addition to the constructs defined in the *Strict* model, special predicates are provided on `float` and `double` numbers, which check that their arguments are finite numbers, NaNs or infinities respectively:

```
predicate \is_NaN(float x);
predicate \is_NaN(double x);
predicate \is_finite(float x);
predicate \is_finite(double x);
predicate \is_infinite(float x);
predicate \is_infinite(double x);
predicate \is_minus_infinity(float x);
predicate \is_minus_infinity(double x);
predicate \is_plus_infinity(float x);
predicate \is_plus_infinity(double x);
```

Useful predicates are also available to express the comparison operators of `float` (resp. `double`) numbers as in C: `\le_float`, `\lt_float`, `\ge_float`, `\gt_float`, `\eq_float`, `\ne_float` (resp. for `float` and `double` numbers). They are translated to the operators `float_le_float`, etc ... defined in the *Why* side.

The sign of a floating-point number is given by the built-in logic function:

```
logic sign \sign(float x);
logic sign \sign(double);
```

where `sign` is an ACSL built-in logic type with two elements `\Positive` and `\Negative`.

Casting a too large `real` may result into one of the special values $\pm\infty$, $\pm$`max_float` and $\pm$`max_double` accroding to the current `rounding_mode` where `max_float` and `max_double` are respectively the maximal floating-point numbers represented in the `float` and `double` formats.

4.2.2 A coq formalization of the Full model

With the *Full* model, the *Coq* implementation of the `gen_float` type is a seven fields record composed of:

- the field `genf` of binary floating-points `float2` of the *Gappa* library;
- the `Float_class` field;
- the `sign` field;
- a `sign_invariant` to satisfy the `finite_sign` axiom as above;
- the floating-point value defined by: `float_value := float2R genf`;
- an exact value of type `R`;
- a model value of type `R`.

4.3 Examples

In this subsection, we illustrate the efficiently of our models on several short floating-point C programs. Pragmas allow the user to switch from one model to another. There are three pragmas: the first one is the default model `real`, the pragma `JessieFloatModel(strict)` for the *Strict* model and `JessieFloatModel(full)` for the *Full* model. There are also five other pragmas for the five rounding directions of the IEEE standard: the default pragma is `#pragma JessieFloatRoundingMode(nearest)` and the pragmas:

```
#pragma JessieFloatRoundingMode(downward)
#pragma JessieFloatRoundingMode(upward)
#pragma JessieFloatRoundingMode(towardzero)
#pragma JessieFloatRoundingMode(towardawayzero)
```

4.3.1 Polynomial approximation of transcendental functions

The cosine function:

The following annotated C program implements a *Taylor* polynomial approximation of order 2 of the cosine function in a neighborhood of zero.

```
#pragma JessieFloatModel(strict)

/*@ requires \abs(x) <= 0x1p-5;
  @ ensures \abs(\result - \cos(x)) <= 0x1p-23;
  @*/
float my_cosine(float x) {
  //@ assert \abs(1.0 - x*x*0.5 - \cos(x)) <= 0x1p-24;
  return 1.0f - x * x * 0.5f;
}
```

The exponential function:

The following annotated C program implements a polynomial approximation of order 2 of the exponential function in a neighborhood of zero by *Remez* algorithm (the *minmax* approximation).

```
#pragma JessieFloatModel(strict)

/*@ requires \abs(x) <= 1.0;
  @ ensures \abs(\result − \exp(x)) <= 0x1p−4;
  @*/

double my_exp(double x) {
  /*@ assert \abs(0.9890365552 + 1.130258690*x + 0.5540440796*x*x
    @          − \exp(x)) <= 0x0.FFFFp−4;
    @*/
  return 0.9890365552 + 1.130258690 * x + 0.5540440796 * x * x;
}
```

When compiling these two annotated programs by *Frama-C*, the proof obligations generated by *Why* are partially proven by automatic theorem provers thanks to our axiomatization:

```
axiom bounded_real_no_overflow :
  forall f:float_format. forall m:mode. forall x:real.
      abs_real(x) <= max_gen_float(f) −> no_overflow(f,m,x)
```

Gappa tool proves all these proof obligations except the two assertions which are proven by Coq using the tactic *interval* [15] from the *Gappa* library.

4.3.2 Interval arithmetic

The following annotated C program implements arithmetic floating-point operations on intervals with floating-point bounds. The current rounding mode in this program is `Down`.

```
#pragma JessieFloatModel(full)
#pragma JessieFloatRoundingMode(downward)

/*@ predicate dif_sign(double x, double y) =
  @    \sign(x) != \sign(y);
  @ predicate sam_sign(double x, double y) =
  @    \sign(x) == \sign(y);
  @ predicate double_le_real(double x,real y) =
  @    (\is_finite(x) && x <= y) || \is_minus_infinity(x);
  @ predicate real_le_double(real x,double y) =
  @    (\is_finite(y) && x <= y) || \is_plus_infinity(y);
  @ predicate in_interval(real a,double l,double u) =
  @    double_le_real(l,a) && real_le_double(a,u);
  @ predicate is_interval(double xl, double xu) =
  @    (\is_finite(xl) || \is_minus_infinity(xl))
```

```
  @     &&
  @    (\is_finite(xu) || \is_plus_infinity(xu));
  @*/

double zl, zu;

/*@ requires is_interval(xl,xu) && is_interval(yl,yu);
  @ ensures is_interval(zl,zu);
  @ ensures \forall real a,b;
  @    in_interval(a,xl,xu) && in_interval(b,yl,yu) ==>
  @    in_interval(a+b,zl,zu);
  @*/
void add(double xl, double xu, double yl, double yu)
{
  zl = xl + yl;
  zu = -(-xu - yu);
}

/*@ requires ! \is_NaN(x) && ! \is_NaN(y)
  @    && (\is_infinite(x) ==> y != 0.0 && dif_sign(x,y))
  @    && (\is_infinite(y) ==> x != 0.0 && dif_sign(x,y));
  @ assigns \nothing;
  @ ensures double_le_real(\result,x*y);
  @*/
double mul_dn(double x, double y)
{
  return x * y;
}

/*@ requires !\is_NaN(x) && !\is_NaN(y) && sam_sign(x,y) &&
  @    (\is_infinite(x) ==> y != 0.0 && \abs(y) >= 0x2.0p-1074 )
  @    &&
  @    (\is_infinite(y) ==> x != 0.0 )
  @    &&
  @    (\is_finite(y) && !\no_overflow(\Double,\Down,-y) &&
  @           \sign(y) == \Positive ==> x != 0.0 );
  @ ensures  real_le_double(x * y,\result);
  @*/
double mul_up(double x, double y)
{
  return -(x * -y);
}

/*@ requires !\is_NaN(x) && !\is_NaN(y);
  @ ensures \le_float(\result,x) && \le_float(\result,y);
  @ ensures \eq_float(\result,x) || \eq_float(\result,y);
  @*/
double min(double x, double y)
{
  return x < y ? x : y;
```

```
}

/*@ requires !\is_NaN(x) && !\is_NaN(y);
  @ ensures \le_float(x,\result) && \le_float(y,\result);
  @ ensures \eq_float(\result,x) || \eq_float(\result,y);
  @*/
double max(double x, double y)
{
  return x > y ? x : y;
}
```

When compiling this annotated program by *Frama-C*, the proof obligations generated by *Why* are completely proven by automatic theorem provers thanks to our axiomatization of the two *Why* models:

```
axiom round_increasing: forall f:float_format. forall m:mode.
  forall x:real. forall y:real.
  x <= y -> round_float(f,m,x) <= round_float(f,m,y)

axiom round_down_le: forall f:float_format. forall x:real.
  round_float(f,down,x) <= x

axiom round_up_ge: forall f:float_format. forall x:real.
  round_float(f,up,x) >= x

axiom round_down_neg: forall f:float_format. forall x:real.
  round_float(f,down,-x) = -round_float(f,up,x)

axiom round_up_neg: forall f:float_format. forall x:real.
  round_float(f,up,-x) = -round_float(f,down,x)

axiom round_idempotent: forall f:float_format. forall m1:mode.
  forall m2:mode. forall x:real.
  round_float(f,m1,round_float(f,m2,x)) = round_float(f,m2,x)
```

5 Conclusions and perspectives

In this paper, we have presented two formal models to deal with floating-point C programs: the *Strict* and the *Full* models. Also, an extention of ACSL and a Coq formalization are given according to these two models.

We have seen in Section 4.3 that automatic theorem provers are able to prove the correctness of some floating-point interval arithmetic by using a fruitful axiomatization of the models. Proofs become very long and very complicated when we try to use interactive proof assistants. For example, the proof of the `add` function of intervals in Coq contains 16 cases to study separetely (for different classes of arguments). Our next goal is to use automatic theorem provers as tactics inside Coq to simplify the proof obligations. A more complicated function is the following multiplication function of intervals (which calls the functions `mul_dn`, `mul_up`, `min` and `max` from Section 4.3):

```
double zl,zu;
/*@ requires is_interval(xl,xu) && is_interval(yl,yu);
  @ ensures is_interval(zl,zu);
  @ ensures \forall real a,b;
  @    in_interval(a,xl,xu) && in_interval(b,yl,yu) ==>
  @    in_interval(a*b,zl,zu);
  @*/
void mul(double xl, double xu, double yl, double yu)
{
  if (xl < 0.0)
    if (xu > 0.0)
      if (yl < 0.0)
        if (yu > 0.0) // M * M
          { zl = min(mul_dn(xl, yu), mul_dn(xu, yl));
            zu = max(mul_up(xl, yl), mul_up(xu, yu)); }
        else          // M * N
          { zl = mul_dn(xu, yl);
            zu = mul_up(xl, yl); }
      else
        if (yu > 0.0) // M * P
          { zl = mul_dn(xl, yu);
            zu = mul_up(xu, yu); }
        else          // M * Z
          { zl = 0.0;
            zu = 0.0; }
    else
      if (yl < 0.0)
        if (yu > 0.0) // N * M
          { zl = mul_dn(xl, yu);
            zu = mul_up(xl, yl); }
        else          // N * N
          { zl = mul_dn(xu, yu);
            zu = mul_up(xl, yl); }
      else
        if (yu > 0.0) // N * P
```

```
          { zl = mul_dn(xl, yu);
            zu = mul_up(xu, yl); }
        else            // N * Z
          { zl = 0.0;
            zu = 0.0; }
  else
    if (xu > 0.0)
      if (yl < 0.0)
        if (yu > 0.0) // P * M
          { zl = mul_dn(xu, yl);
            zu = mul_up(xu, yu); }
        else            // P * N
          { zl = mul_dn(xu, yl);
            zu = mul_up(xl, yu); }
      else
        if (yu > 0.0) // P * P
          { zl = mul_dn(xl, yl);
            zu = mul_up(xu, yu); }
        else            // P * Z
          { zl = 0.0;
            zu = 0.0; }
    else                // Z * ?
      { zl = 0.0;
        zu = 0.0; }
}
```

6 Technical appendix

In order to prove the above floating-point C programs, one needs to compile and install the following tools and libraries:

- The SVN version of the *Frama-C* tool [1].
- The CVS version of the *Why* platform [6].
- The last version of *Coq* (*Coq 8.2*) available in the new official website of the *Coq* proof assistant: `http://logical.saclay.inria.fr/coq/?q=node/60`.
- The *Gappa* tool [14] with its libraries: *gappalib-coq-0.9* (for the *gappa* tactic) and *interval-0.8* [15] (for the *interval* and `interval_intro` tactics).
- The *Coq* library on Floating-Point Arithmetic (PFF) which is avalaible on the website: `http://lipforge.ens-lyon.fr/www/pff/`
- The *Coq* module *JessieGappa.v* to translate *Jessie* goals into *Gappa* goals.

References

[1] Frama-C: Framework for modular analyses of C. `http://www.frama-c.cea.fr/`.

[2] IEEE standard for floating-point arithmetic. IEEE-754-2008.

[3] *The Coq proof assistant, Calling external provers*, chapter 25. INRIA, 2008.

[4] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*, 2008. `http://frama-c.cea.fr/acsl.html`.

[5] Sylvie Boldo and Jean-Christophe Filliâtre. Formal Verification of Floating-Point Programs. In *18th IEEE International Symposium on Computer Arithmetic*, pages 187–194, Montpellier, France, June 2007.

[6] Jean-Christophe Filliâtre. The Why verification tool, 2002. `http://why.lri.fr/`.

[7] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In Jim Davies, Wolfram Schulte, and Mike Barnett, editors, *6th International Conference on Formal Engineering Methods*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29, Seattle, WA, USA, November 2004. Springer.

[8] Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. pages 173–177. 2007.

[9] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969.

[11] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system release 3.01, Documentation and user's manual*, March 2001. `http://www.caml.org/ocaml/htmlman/index.html`.

[12] Laurent Théry Marc Daumas, Laurence Rideau. A generic library for floating-point numbers and its application to exact computing. *TPHOLs*, 2001.

[13] Claude Marché. Jessie: an intermediate language for Java and C verification. In *PLPV '07: Proceedings of the 2007 workshop on Programming Languages meets Program Verification*, pages 1–2, Freiburg, Germany, 2007. ACM.

[14] Guillaume Melquiond. Gappa : Automatic proof generation of arithmetic properties. `http://lipforge.ens-lyon.fr/www/gappa/`.

[15] Guillaume Melquiond. Interval package for coq, `http://www.msr-inria.inria.fr/~gmelquio/soft/coq-interval/`.

[16] Guillaume Melquiond. Floating-point arithmetic in the Coq system. In *Proceedings of the 8th Conference on Real Numbers and Computers*, pages 93–102, Santiago de Compostela, Spain, 2008.

[17] Yannick MOY. Jessie plugin tutorial. 2008. `http://frama-c.cea.fr/jessie.html`.

[18] Yannick MOY. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, University Paris XI, 2009.

[19] Virgile Prevosto. ACSL mini-tutorial. 2008.

[20] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.2*, July 2008. `http://coq.inria.fr`.

Contents

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 Orsay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399